JOURNAL OF COMPUTER AND SYSTEM SCIENCES 44, 103–113 (1992)

Processor-Time Tradeoffs in PRAM Simulations

PRABHAKAR RAGDE

Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Received May 28, 1989; revised March 1, 1990

Different models of concurrent-read, concurrent-write parallel random access machine (CRCW PRAM) are distinguished by their method of write-conflict resolution, which can affect the power of the model. We consider the situation where a weaker model with kn processors wishes to simulate a stronger model with n processors (k may be a function of n). In the case of COMMON simulating ARBITRARY or PRIORITY, we show that one step of the stronger model can be simulated by T steps of the weaker model, where T satisfies the tradeoff $kT \log T = O(\log n)$. In the case of ARBITRARY simulating PRIORITY, we can achieve the tradeoff $T \log(k+1) = O(\log n)$. This improves the number of processors necessary to achieve constant time to $n(\log n)^{\varepsilon}$ (for fixed $\varepsilon > 0$). These tradeoffs unify and extend many previous results in this area. Corresponding lower bounds are proved for the class of simulations in which k processors of the weaker model are assigned to simulate one processor in the stronger model. Two different methods are used to prove the lower bounds; both have a strong combinatorial flavor.

1. INTRODUCTION

The shared-memory parallel RAM is a natural and widely used model of parallel computation. In such a machine, some number of processors $P_1, P_2, \ldots$ communicate by means of synchronized reads and writes to cells of shared memory. Each step of computation consists of three phases. In the read phase, each processor may choose a cell of shared memory from which it reads. All reads take place simultaneously. In the compute phase, an arbitrary amount of local computation is allowed. This rather unrealistic assumption is made only to ensure that lower bounds are robust with respect to choice of processor instruction set; in practice, algorithms usually do a "reasonable" amount of local computation at each step. Finally, in the write phase, each processor may choose a cell of shared memory into which it writes; all writes take place simultaneously. Concurrent access by several processors to the same cell is permitted, and arbitrary values may be written.

Since several processors may simultaneously attempt to write different values into

0022-0000/92 $3.00

the same memory cell, a method of write-conflict resolution must be specified. Here are three models which have appeared in the literature.

COMMON [K]: Simultaneous writes of different values are not allowed; whenever several processors simultaneously write to the same cell, they must be writing the same value.

ARBITRARY [SV1]: When a simultaneous write occurs, one of the values being written appears in the cell, but it is impossible to tell in advance which value will appear. From the point of view of worst-case upper and lower bounds, one may assume that an adversary chooses the winner of each competition so as to make the algorithm take as long as possible.

PRIORITY [Go]: Among the processors simultaneously writing into a cell, the processor of lowest index succeeds.

These models are presented in non-decreasing order of power. Since an algorithm may take advantage of a particular method of write-conflict resolution [SV2, Go, TV], it is important to know if algorithms may be run on weaker models without time penalty. In [FRW1], the conjecture is made that there is a $\Theta(\log n)$ separation between any two of these models with the same number of processors and memory cells. Partial progress has been made toward resolving this conjecture.

This paper contributes results that unify and extend many of the previous results (described below) by considering general tradeoffs between the number of processors in the weaker model and the time required for simulation. In Section 2, we show that one step of PRIORITY with n processors can be simulated by T steps of COMMON with kn processors, where $kT \log T = O(\log n)$. Another way of writing this is that $T = O(\log n/k(\log\log n - \log k))$.

We also show that one step of PRIORITY with n processors can be simulated by T steps of ARBITRARY with kn processors, where $T = O(\log\log n/\log(k+1))$. When $k = (\log n)^\varepsilon$ for any fixed $\varepsilon > 0$, we see that ARBITRARY with $n(\log n)^\varepsilon$ processors can simulate PRIORITY with n processors in constant time. No constant-time simulation using this few processors was previously known.

In Section 3 we show corresponding lower bounds which prove these tradeoffs optimal with respect to a wide class of simulations, namely those in which k processors are assigned to simulate one processor and work only on its write conflicts. The fact that these models are equivalent modulo a polynomial blowup in the number of processors and memory cells means that lower bound techniques must be very sensitive. While we cannot show that the lower bounds work for general simulation, it should be noted that all bounds separating these models are restricted in some fashion—by limiting the size of shared memory or by requiring large inputs. Two different lower bound techniques are used, and the approaches taken indicate that these techniques might find applications in the theory of fault-tolerant PRAMs.

The conjecture that there is a $\Theta(\log n)$ separation between any two of these

models with the same number of processors and memory cells has been shown when the number of memory cells is $O(n^\varepsilon)$ for fixed $\varepsilon > 0$ [FRW2], and a separation of $\Omega(\sqrt{\log n})$ between COMMON and ARBITRARY was demonstrated in the case of infinite memory [RSSW], improved by Boppana [Bo] to $\Omega(\log n/\log\log n)$. Reference [GR] shows separations between related models with infinite memory. These last three results require very large input values.

Reference [FRW1] showed a simple simulation of one step of any model by $O(\log n)$ steps of any other model. Reference [FRW3] showed that if the weaker model is allowed to use more memory cells, then one step of PRIORITY can be simulated by $O(\log n/\log\log n)$ steps of COMMON (thus Boppana's result is optimal); this is the case $k = 1$ in our simulation of PRIORITY by COMMON. Reference [CDHR] showed that only $O(\log\log n)$ steps of ARBITRARY are necessary to simulate one step of COMMON; this is the case $k = 1$ of our simulation of ARBITRARY by COMMON.

A related approach allows more processors in the weaker model, and concentrates on getting constant-time simulations. Reference [K] showed that $O(n^2)$ processors of COMMON suffice to simulate one step of PRIORITY in constant time; [FRW1] improved that to $O(n^{1+\varepsilon})$. Reference [CDHR] showed that $O(n \log n)$ processors suffice; this is the case $k = O(\log n)$ in our simulation of PRIORITY by COMMON. The authors of [CDHR] independently obtained this result for the case $k = (\log n \log\log\log n)/(\log\log n)$ [H]; our general result is implicit in their approach. Boppana [B] has independently obtained the same bounds for simulating PRIORITY by COMMON.

2. UPPER BOUNDS

We use the approach first taken in [FRW3] and used further in [CDHR] and [HN]. Assign k processors (call them $P_{i,1}, P_{i,2}, \ldots P_{i,k}$) in the weaker model to simulate processor P_i in the PRIORITY model. Consider the write phase of a step in a PRIORITY algorithm. Each simulated processor P_i has a cell into which it wishes to write. A natural allocation of labor is to have each of the processors simulating P_i work only on resolving the conflict at that cell. Once the identity of the winner is known, the write can take place without conflict. Each cell in the simulated model can be assigned some auxiliary memory in the simulating model, to be used in determining the winner for that cell. The difficulty for a set of k processors is knowing which other sets of k are working on the same problem. Those working on different problems will appear "dead" by not participating in their problem. This leads naturally to the following problem definition:

k-set LEFTMOST PRISONER

Before: There are n sets of k processors, with indices 1 through n. The processors in set i are labeled $P_{i,1}$ through $P_{i,k}$. Each set is either live or dead. Dead sets do not participate.

After: Each set knows whether or not it is the live set of lowest index (the "winner").

A T-step solution to k-set LEFTMOST PRISONER on a weaker model leads to a simulation of one step of PRIORITY with n processors by T steps of the weaker model with kn processors. This paper presents new upper and lower bounds for the k-set LEFTMOST PRISONER problem. A "prisoner" PRAM (one in which part of the input specifies which processors participate in the solution) is called an "allocated" PRAM in [HN] and a "sack of processors" in [Gr].

THEOREM 1. *A k-set LEFTMOST PRISONER problem involving n sets of processors can be solved on COMMON in T steps, where $kT \log T = O(\log n)$.*

Proof. To solve LEFTMOST PRISONER on COMMON, we give each set of k processors a code. The code of set i is $i-1$ expressed as a k-digit number in base $\lceil n^{1/k} \rceil$. Suppose set i has code $d_1 d_2 \cdots d_k$. Then processor $P_{i,j}$ will represent the group of sets whose codes begin with $d_1 d_2 \cdots d_j$ and will work on the subproblem of determining the group containing the live set of lowest index among the $\lceil n^{1/k} \rceil$ groups whose codes start with $d_1 d_2 \cdots d_{j-1}$. We call this the subproblem at level j. $P_{i,j}$ wins this subproblem (along with every live processor in its group) if there is no live set whose code starts with $d_1 d_2 \cdots d_{j-1} q$, for any $q < d_j$.

It is not hard to see that set i is the live set of lowest index if and only if each processor in set i wins its subproblem; that is, set i is the winner of its subproblems at all levels. As an example, let $n = 27$ and $k = 3$. A code is a three-digit number; set 6 has code 012. The group 01* in the subproblem at level 2 consists of the sets with codes 010, 011, and 012 (sets 4, 5, 6); if set 6 is alive and sets 1 through 3 are all dead, this group will win its subproblem at level 2.

Since every processor is assigned one subproblem, all subproblems can be done in parallel, and it takes only one more step for a set to check whether or not all its members won their subproblems (this is just a k-way OR). Each subproblem constitutes an instance of 1-set LEFTMOST PRISONER; there may be several representatives for each group, but they will all act the same, so it is as if one processor represented that group. In [FRW3], it is shown that a 1-set LEFTMOST PRISONER of size s can be solved on COMMON in time $T = O(\log s / \log \log s)$. Here $s = \lceil n^{1/k} \rceil$, yielding $kT \log T = O(\log n)$. (When s is small, letting $s = 2$ suffices; T will be constant.) A careful analysis shows that the amount of memory used is $O(n)$.

THEOREM 2. *A k-set LEFTMOST PRISONER problem involving n sets of processors can be solved on ARBITRARY in T steps, where $T \log(k+1) = O(\log \log n)$.*

Proof. To obtain the improved tradeoff for solving k-set LEFTMOST PRISONER on ARBITRARY, we use a similar approach, except that recursion is introduced. This time, the code of a set of k processors with index i is $i-1$ expressed as a $k+1$-digit number in base $\lceil n^{1/k+1} \rceil$. As before, if the code of set i

is $d_1 d_2 \cdots d_{k+1}$, then there are $k+1$ subproblems in which set i could participate. The subproblem at level j, from the point of view of set i, is whether the group of sets whose codes begin with $d_1 d_2 \cdots d_i$ has the live set of lowest index among those groups of sets whose codes begin with $d_1 d_2 \cdots d_{i-1}$. Set i is the live set of lowest index if and only if it is the winner of its subproblems on all $k+1$ levels.

Because we use recursion, however, all processors in a set must be devoted to the same subproblem. We use the fact that "elections" can be done in constant time on the ARBITRARY model, as explained below. We assign $\lceil n^{1/k+1} \rceil$ cells of auxiliary memory to each subproblem, one cell for each group of sets involved in the subproblem. At the first step, $P_{i,j}$ from set i attempts to write i into the cell associated with its group in its subproblem at level j. Processors from other sets will be vying to be elected representative of the same group of sets (those whose codes begin with $d_1 d_2 \cdots d_j$); the winner is the one that succeeds in writing.

If none of the processors in set i succeed (this condition can be checked in one step), then they assign themselves to the subproblem at level $k+1$. If only one processor succeeds, say $P_{i,j}$, then they assign themselves to the subproblem at level j. However, if two or more processors in set i succeed in writing their names, then set i assigns itself to the subproblem at the level of lowest index in which one of its processors succeeded. By using the LEFTMOST ONE algorithm of [FRW1], this can be determined within set i in $O(1)$ steps.

Each set is now assigned to a subproblem of size $\lceil n^{1/k+1} \rceil$; recursively, all subproblems are solved in parallel. The winner of each subproblem is posted. Some subproblems may not have any processors assigned to them; it is understood that they have no posted winners. Set i knows whether or not it is the winner in the subproblem that it worked on; it must find out about the other k subproblems. One processor from the set is assigned to go find out about each subproblem. If no winner is posted in a subproblem for i, then i may assume that it is the winner.

But the posted winner of a subproblem may not be the actual winner. Suppose that processors from sets i and j, for $i<j$, attempted to write into different cells during elections in the same subproblem, one at level q. They may both have been elected, but set i may have gone off to solve a subproblem at level p, for $p<q$, leaving its group at level q unrepresented. In this case the group of which set j is the representative may think it has won the problem at level q.

If this has happened, then $P_{j,p}$ knows about it. The codes of set i and j agree in the first $q-1$ positions, and hence they are in the same group on level p. This means that both $P_{j,p}$ and $P_{i,p}$ attempted to write into the same cell; $P_{i,p}$ succeeded, and $P_{j,p}$ learned that it failed, by seeing the value i in the cell.

Thus the condition that set i must achieve after the recursion is completed is that its group must be the posted winner in all of its subproblems, and that no processor in the set has seen the name of a set of smaller index. This can be checked in one step. It follows that the time to solve a k-set LEFTMOST PRISONER of size n on ARBITRARY is the time needed to solve a problem of size $\lceil n^{1/k+1} \rceil$ plus a constant. This recurrence yields the tradeoff $T \log(k+1) = O(\log \log n)$, as required. Again, the amount of memory used is $O(n)$.

3. Lower Bounds.

The lower bounds we prove here are for the k-set LEFTMOST PRISONER problem. This means that the tradeoffs discussed in the previous section are optimal among all simulations that have the following form: k processors on the simulating machine are assigned to one processor on the simulated machine, and those k processors work only on the task of resolving the conflicts at the cells that the simulated processor wishes to write into. This is a restriction on the communication between processors; the simulations in Section 2 are all of this form. It is not clear whether the bounds hold for general simulations. They are, however, independent of the number of memory cells on the simulating machine. Boppana [Bo] has shown that discovering whether k integers are all distinct on COMMON with n processors requires time T, where $kT \log T = \Omega(\log n)$, provided that the integers can be chosen from a large range. This problem can be solved in $O(1)$ time on ARBITRARY by using memory as a hash table; it follows that the simulation of ARBITRARY by COMMON given in Section 2 is optimal when both machines have very large memories. The possibility remains, however, that more efficient simulations can be found when the number of shared memory cells is polynomial in the number of processors.

To prove our lower bounds, we strengthen the machine model. Instead of having kn processors, we have n processors, but give each the ability to read and write k cells per step. We call this a k-access PRAM. Any algorithm for the k-set LEFTMOST PRISONER problem leads to an algorithm for 1-set LEFTMOST PRISONER on a k-access PRAM. Hence it suffices to prove a lower bound for 1-set LEFTMOST PRISONER on a k-access PRAM. Since the very first read phase is useless, we also assume that a step consists of write, read, and compute phases, in that order. This makes a difference of at most one in the number of steps an algorithm takes.

To prove the lower bound for the COMMON model, we extend the approach of [FRW3]. The key to the lower bound is the fact that, in the COMMON model, if a processor writes into a cell it cannot tell which other processors wrote into that cell at the same time. Hence it is difficult for a live processor to tell if another live processor exists.

THEOREM 3. *Any algorithm to solve a 1-set LEFTMOST PRISONER of size n on k-access COMMON requires T steps, where $kT \log T = \Omega(\log n)$.*

Proof. Consider a weaker prisoner-style problem, which we call WEAK THRESHOLD-2. Initially, each processor is alive or dead; after the problem is solved, there must be a processor in state 1 if two or more processors are alive. If only one processor is alive, it must be in state 0.

Any algorithm for 1-set LEFTMOST PRISONER leads to an algorithm for WEAK THRESHOLD-2, since any live processor that is not the live processor of lowest index just goes into state 1. Furthermore, because of the weak termination condition of WEAK THRESHOLD-2, we can assume that processors access

memory in a semi-oblivious fashion. That is, for any processor P and any time-step t, the set of k cells that P could read or write into at step t does not depend on the input. The reason for this is that as soon as a processor reads a value that it did not write at a previous step, it can halt in state 1. Before this occurs, the only information it has is that it is alive.

An algorithm for WEAK THRESHOLD-2 thus yields a set $\mathcal{S}$ of n sequences, one for each processor. A sequence for a processor is of length $2T$, and its entries are k-sets of cell indices. Entry $2t-1$ specifies the set of k cells read by the processor at step t, and entry $2t$ specifies the cells into which the processor writes at step t. This set of sequences $\mathcal{S}$ has an important property, which we call the difference property: for any two sequences α and β, there exist i, j, d such that d is in one of α_i, β_i, but not both. Furthermore, if $d \in \alpha_i$, then $d \in \beta_j$, and if $d \in \beta_i$, then $d \in \alpha_j$. We say that the difference property is witnessed by positions i, j, and value d.

If this were not true for the sequences of processors P_i and P_j, for $i<j$, then those processors would access the same cells at each step. Thus P_j cannot tell the difference between the case when it is the only live processor (in which case it should halt in state 0), and when P_i is the only other live processor. It must halt in state 0 for both these cases; but P_j has the same problem, and when both are alive, they will both halt in state 0.

We will show that the number of sequences of length $2T$ in $\mathcal{S}$ is bounded above by $(2T)^{2kT}$. Since $n \leqslant (2T)^{2kT}$, the lower bound follows. The following technical lemma shows that we can assume, for the purposes of bounding the size of the set, that within any sequence, any two set-entries are disjoint. This means that for solving WEAK THRESHOLD-2, no processor need access the same cell more than once—a fact that seems intuitively obvious, but whose proof is not particularly elegant.

LEMMA 4. *If there exists a set $\mathcal{S}$ of n sequences of length $2T$ with the difference property, then there exists a set $\mathcal{S}'$ of n sequences of length $2T$ with the difference property such that for any $\alpha \in S$, any $0 \leqslant i < j \leqslant 2T$, $\alpha_i \cap \alpha_j = \varnothing$.*

Proof. Let $\mathcal{S}$ be a set of n sequences of length $2T$ which maximizes $r = \max\{k \mid \forall \alpha \in \mathcal{S}, \forall 1 \leqslant i < j \leqslant k, \alpha_i \cap \alpha_j = \varnothing\}$. That is, $\mathcal{S}$ is a set of sequences such that the set-entries of any two sequences are disjoint up to and including position r, and r is as large as possible. If $r = 2T$, the theorem follows. Otherwise, we construct a set $\mathcal{S}'$ of sequences that contradicts the choice of $\mathcal{S}$.

Let $R(\alpha, i) = \bigcup\{\alpha_i \cap \alpha_j \mid j < i\}$, for $\alpha \in \mathcal{S}$, $0 \leqslant i \leqslant 2T$. $R(\alpha, i)$ is the set of cell indices that are repeated in position i of sequence α. Let $R = \bigcup\{R(\alpha, i) \mid \alpha \in \mathcal{S}, 0 \leqslant i \leqslant 2T\}$. R is the set of indices that are mentioned more than once in some sequence. Let $f: R \to E$ be any one-to-one function, where $E \cap \alpha_i = \varnothing$ for any $\alpha \in \mathcal{S}$, $0 \leqslant i \leqslant 2T$. Extend f to subsets of R in the natural fashion. Let $\mathcal{S}'$ be constructed as follows: for each $\alpha \in \mathcal{S}$, let $\alpha' \in \mathcal{S}'$ satisfy $\alpha'_i = \alpha_i \backslash R(\alpha, i) \cup f(R(\alpha, i))$.

By construction, $\alpha'_i \cap \alpha'_j = \varnothing$ for any $\alpha' \in \mathcal{S}'$, $0 \leqslant i < j \leqslant r+1$. It is not hard to show that $\mathcal{S}'$ has the difference property, contradicting the choice of $\mathcal{S}$. Given

sequences $\alpha, \beta \in \mathcal{S}$, let α', β' be the corresponding sequences in $\mathcal{S}'$. Since $\mathcal{S}$ has the difference property, without loss of generality there exists $i<j$ and d such that $d \in \alpha_i$, $d \notin \beta_i$, $d \in \beta_j$. If this is not true for d in α' and β', then there are three possibilities.

The first possibility is that $d \notin \beta'_i$. In this case, $f(d) \in \alpha'_i$ and $f(d) \in \beta'_j$. Since $d \notin \beta_i$, $f(d) \notin \beta'_i$. Thus the difference property holds for α', β', as witnessed by positions i, j, and value $f(d)$.

The second possibility is that $d \in \beta'_j$ but $d \notin \alpha'_i$. In this case, $d \in R(\alpha, i)$; let k be the smallest integer such that $d \in \alpha_k$. It follows that $k<i$. It must be that $d \notin \beta'_k$, for if $d \in \beta'_k$, then $d \in R(\beta, j)$, contradicting the fact $d \in \beta'_j$. So the difference property holds for α', β' as witnessed by positions k, j, and value d.

Finally, it could be that $d \in \alpha'_i$ but $d \notin \beta'_j$. Then $f(d) \in \beta'_j$. Let k be the smallest integer such that $d \in \beta_k$; as before, $k<j$. If $k<i$, then $d \notin \alpha'_k$ because $d \in \alpha'_i$. In this case, positions k, i with value d are witnesses. If $k>i$, then position i, k with value d are witnesses.

Let c be an integer such that all cell indices appearing in sequences of $\mathcal{S}$ are in the set $\{1, 2, \ldots c\}$. Consider the set of ordered tuples of length c with entries chosen from $\{1, 2, \ldots 2T\}$. Obviously, there are $(2T)^c$ such tuples. We say a tuple A is consistent with a sequence α in $\mathcal{S}$ if, for all cell indices j, $j \in \alpha_i$ implies $A_j = i$. Intuitively, A spells out the time at which each cell mentioned in α is accessed. We can assume that the entries of A are single indices because of Lemma 4. Entries in A corresponding to cells not in α can have any value. Since fixing α only fixes at most $2kT$ positions of a consistent tuple A, it follows that there are at least $(2T)^{c-2kT}$ tuples consistent with any sequence $\alpha \in \mathcal{S}$.

It is also true that no tuple A is consistent with two different sequences in $\mathcal{S}$. For any two sequences α, β, there must be a cell index i such that $i \in \alpha_j$ and $i \in \beta_k$, for $k \neq j$. But then A_i would have to be both j and k, which is impossible.

Thus the union, over all sequences $\alpha \in \mathcal{S}$, of the set of tuples consistent with α is a disjoint union. If there are n sequences, there are at least $n(2T)^{c-2kT}$ tuples in this union; but there are at most $(2T)^c$ tuples. It follows that $(2T)^{2kT} \geqslant n$.

A similar proof will not work on ARBITRARY, since WEAK THRESHOLD-2 can be solved in $O(1)$ steps. We prove a lower bound on LEFTMOST PRISONER by an adversary argument. This technique was used in [Gr] to show that p steps are required (for small p) to compute the parity of the number of live processors on PRIORITY when at most p processors are known to be alive. The proof of our bound for $k=1$ was inherent in this development.

THEOREM 5. *Any algorithm to solve a* 1*-set LEFTMOST PRISONER of size n on k-access ARBITRARY requires T steps, where* $T \log(k+1) = \Omega(\log \log n)$.

Proof. Given a k-access ARBITRARY algorithm to solve 1-set LEFTMOST PRISONER, we define a set of inputs I_t that the algorithm cannot distinguish after step t. (An input is just a specification of which processors are live). I_t is defined with the help of three disjoint sets of processor indices A_t, D_t, and V_t, whose union

is $\{1, 2, \ldots n\}$. I_t is the set of all inputs such that all processors in A_t are live and all processors in D_t are dead. We are also allowed to specify, for each write conflict that occurs on an input in I_t, exactly which processor wins each competition to write.

In order to define the notion of "distinguishing" two inputs, we must define what it means for a processor to detect the liveness of another processor. Processor P detects processor Q on I_t if there exist two inputs in I_t, differing only in the liveness of Q, such that the state of P after step t differs on the two inputs. Similarly, we may speak of a cell detecting processor Q on I_t.

By construction, we maintain four conditions on I_t:

(i) No processor detects any other processor.

(ii) Each cell detects at most one processor.

(iii) Each index in A_t is greater than every index in V_t.

(iv) $|A_t| \leqslant t$.

Suppose the final set V_T is non-empty. Consider two inputs in I_T, one in which all processors in V_T are dead, and one in which some processor P_i (for $i \in V_T$) is alive. The winning processor for the first input cannot detect the liveness of P_i after step T, and so cannot distinguish the two inputs. Since P_i is the winning processor in the second input, the algorithm is faulty. We bound the size of V_t from below as a function of t, thus bounding T from below.

Initially, $A_0 = D_0 = \varnothing$, $V_0 = \{1, 2, \ldots n\}$, and all conditions are trivially satisfied. Now suppose we have defined these sets through step t. We must show how to construct A_{t+1} and D_{t+1}. To begin with, we let $V_{t+1} = V_t$, $A_{t+1} = A_t$, and $D_{t+1} = D_t$. If there is some processor in A_t that always attempts to write into a cell at step $t+1$ for all inputs in I_t, then let it always succeed (if several attempts, pick one). For the cells that remain, let S_i, for $i \in \{1, 2, \ldots n\}$, be the set of (at most) k cells that P_i writes into at step $t+1$.

A theorem of Erdős and Rado [ER] states that in any family of at least $k!\, s^{k+1}$ (not necessarily different) sets of size at most k, there is a sunflower formed by at least s sets, that is, a collection of s sets whose pairwise intersection is equal to their intersection. Letting $s = \lfloor (1/k)(|V_t|^{1/(k+1)}) \rfloor$, we see that $\{S_i : i \in V_t\}$ contains a sunflower. Remove from V_{t+1} any processor index not corresponding to a set in this sunflower, and add it to D_{t+1}. As a result, cells are either written into by at most one processor from V_{t+1} (if the cell index is in one of the sunflower sets, or none of them) or by all live processors in V_{t+1} (if the cell index is in all sunflower sets). Take the processor of highest index still in V_{t+1}, move it into A_{t+1}, and have it win the competition to write into all cells that are in all sunflower sets. This satisfies conditions (iii) and (iv).

We have two problems remaining: cells may be able to detect two processors (one up to step t, and one that may or may not write into the cell at step $t+1$), and in the read phase of step $t+1$, a processor may read a cell that detects another processor. We cannot prevent processors in A_{t+1} from detecting processors in the

read phase of step $t+1$, except by killing the processors thus detected. Each processor in A_{t+1} can read k cells, each of which can detect at most two processors; moving at most $2k(t+1)$ processors from V_{t+1} to D_{t+1} takes care of this. Define a directed graph whose vertex set is V_{t+1}, with an edge from processor P to processor Q if Q reads a cell at step $t+1$ that could detect P, or if Q writes into a cell (and has a chance of succeeding) at step $t+1$ that could detect P by step t. There are at most k cells read by Q, and each one can detect at most two processors; there are at most k cells written into by Q and each one can detect at most one processor by step t. Hence the indegree of this graph is at most $3k$.

It is a consequence of Turan's Theorem [B, p. 282] that any graph with n vertices and cn edges has an independent set of size at least $n/(2c+1)$. Thus our graph has an independent set of size at most $|V_t|^{1/(k+1)}/(6k+1)(k)-(t+1)$. Remove every processor not in this independent set from V_{t+1} and place them in D_{t+1}. This satisfies conditions (i) and (ii). From the recurrence

$$|V_0| = n$$

$$|V_{t+1}| \geq \frac{|V_t|^{1/(k+1)}}{(6k+1)k} - (t+1)$$

$$|V_T| < 1$$

it is not hard to show that $T = \Omega(\log\log n/\log(k+1))$.

4. Conclusions and Open Problems

The "prisoner" paradigm is a useful one for algorithm design, as it renders trivial the often thorny problem of processor allocation. Here we have used it to obtain efficient simulations, while demonstrating in two different ways that there are limits to the technique. It would be nice to show that these simulations are optimal among all simulations.

This could also be the starting point for explorations into fault-tolerant PRAMs. Many highly parallel algorithms are extremely sensitive to the failure of a processor; we would like upper bounds showing that computation can be carried on under any failure pattern, or conversely, that processor failure can doom any solution to a certain problem. The techniques shown here work when many processors can fail; an interesting setting in which no lower bound techniques are known is the case where only a few processors fail.

References

[Be] C. Berge, "Graphs and Hypergraphs," North-Holland, Amsterdam, 1973.

[Bo] R. Boppana, Optimal separations between concurrent-write parallel machines, *in* "Proceedings, 30th Annual ACM Symposium on Theory of Computing, 1989," pp. 320–326.

[CDHR] B. CHLEBUS, K. DIKS, T. HAGERUP, AND T. RADZIK, Efficient simulations between CRCW PRAMs, *in* "Proceedings, 13th Symposium on the Mathematical Foundations of Computer Science, 1988," pp. 230–239.

[ER] P. ERDŐS AND R. RADO, Intersection theorems for systems of sets, *J. London Math. Soc.* **35** (1960), 85–90.

[FRW1] F. E. FICH, P. RAGDE, AND A. WIGDERSON, Relations between concurrent-write models of parallel computation (preliminary version), *in* "Proceedings, 3rd ACM Symposium on Principles of Distributed Computation, 1984," pp. 179–184.

[FRW2] F. E. FICH, P. RAGDE, AND A. WIGDERSON, Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* **17** (1988), 606–627.

[FRW3] F. E. FICH, P. RAGDE, AND A. WIGDERSON, Simulations among concurrent-write models of parallel computation, *Algorithmica* **3** (1988), 43–51.

[Ga] Z. GALIL, Optimal parallel algorithms for string matching, *in* "Proceedings, ACM Symposium on Theory of Computing, 1984," pp. 240–248.

[Go] L. GOLDSCHLAGER, A unified approach to models of synchronous parallel machines, *J. Assoc. Comput. Mach.* **29** (1982), 1073–1086.

[Gr] V. GROLMUSZ, "Large Parallel Machines Can Be Extremely Slow for Small Problems," manuscript, 1988.

[GR] V. GROLMUSZ AND P. L. RAGDE, Incomparability in parallel computation, *in* "Proceedings, 27th Annual IEEE Symposium on Foundations of Computer Science, 1987," pp. 89–98.

[H] T. HAGERUP, Personal communication.

[HN] T. HAGERUP AND M. NOWAK, "Parallel Retrieval of Scattered Information," Technical Report 14/1988, Universität des Saarlandes.

[K] L. KUCERA, Parallel computation and conflicts in memory access, *Inform. Process. Lett.* **14** (1982), 93–96.

[RSSW] P. RAGDE, W. STEIGER, E. SZEMEREDI, AND A. WIGDERSON, The parallel complexity of element distinctness is $\Omega(\log n)$, *SIAM J. Discrete Math.* **1** (1988), 399–410.

[SV1] Y. SHILOACH AND U. VISHKIN, Finding the maximum, merging, and sorting on parallel models of computation, *J. Algorithms* **2** (1981), 88–102.

[SV2] Y. SHILOACH AND U. VISHKIN, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3** (1982), 57–63.

[TV] R. TARJAN AND U. VISHKIN, Finding biconnected components and computing tree functions in logarithmic parallel time, *in* "Proceedings, 25th Annual ACM Symposium on Foundations of Computer Science, 1984," pp. 12–20.